TOPPERS 活用アイデア・アプリケーション開発コンテスト

部門　：　活用アイデア部門　（アプリケーション開発部門）

作品のタイトル　：　シュリンク版 TOPPERS/SSP と
それを利用した
タミヤラジコン改造 RaspberryPi スマホリモコンカー

作成者　：　アライブビジョンソフトウェア株式会社　髙橋和浩

対象者　：

使用する開発成果物　：　TOPPERS/SSP

目的・狙い

1.アドオン方式のリアルタイムカーネル

シュリンク版 SSP の特徴は、OS 無環境において、システムの構成を大きく変更せずにリアルタイムカーネルを搭載できることをひとつの目的としています。
想定の主なターゲットユーザーは､内作の RTOS を利用しているユーザーまたは OS 無ですでに組込みシステムを開発しているユーザーです。

シュリンク版の搭載は、組込みでの HelloWorld とも言われる LED をタイマ割り込みで点滅するプログラムまでできた前提の環境にアドオンできるものです。
シュリンク版 SSP をさまざまなシステムに活用しているものアプリケーションとして応募しました。

2.リアルタイムカーネルの移植の手間の軽減

BSP(ボードサポートパッケージ）つまりボードへの移植されているか否かがリアルタイム OS の導入の判断材料になる場合があります。シュリンク版は移植確認作業を軽減することができます。
シュリンク版 SSP は割り込み管理、ドライバ、ブート部を本体とは切り離したものとしているため正規版よりもより少ない変更で移植が可能になります。
例えば、RaspberryPi にもこのシュリンク版で移植しています。

また、待ちのない最少セットカーネルの守備範囲を超えて
1)待ち状態の追加
2)C 言語記述ディスパッチャ
を搭載/実現し、通常の待ちのあるリアルタイムカーネル機能と
ディスパッチャを C 言語で記述することでさらに移植性が向上しています。

例えば、RaspberryPi にもこのシュリンク版で移植しています。

アイデア/アプリケーションの概要

シュリンク版 SSP は、以下のアプリケーションおよび機能を実現しました。
1)C 言語記述ディスパッチャ
ITRON のシステムコールで言う ret_int を C で実装
2014 年のアイデアコンテストのアイデア部門での入賞作かと思いますが、
これを実現しました。機種依存性を低くしたため、比較的容易に機種別の移植
が可能になると考えます。

2)待ち状態の追加
構成管理で、待ちのあるもの無いものいずれも公開しています。
待ち状態は、setjmp/longjmp をコンテキストの保存と復元を基本に、
さらに割り込みからのディスパッチ時の保存/復元をほぼ C 言語だけで
実装しました。詳細を後述します。

3)サンプルアプリケーション
RaspberryPi への移植　B,B+,2 いずれも　ただしシングルコア
アプリケーション例　タミヤラジコン改造　スマホリモコンカー
(オープンソースカンファレンス関西 2015 TOPPERS ブースにてデモ)

詳細

1.シュリンク版 SSP の改造過程

TOPPERS/SSP からシュリンク版 SSP への改造過程として、モジュール単位で採用モジュールを絞ることで実現しました。

ドライバ、ブート部、割り込み管理を除くモジュールで構成します。

これに、タイムイベントハンドラのモジュールはそのまま追加するだけで機能追加ができます。

シュリンク版モジュール選択表　別紙 A 参照ください。

2.C 言語記述ディスパッチャ

TOPPERS/SSP の正規版のサービスコール ret_int はアセンブリ言語で記述されています。タスク毎のコンテキストの保存ではなく割り込み処理でのレジスタの保存と復元の部分の復元にアセンブリ言語が不可欠だからだからです。しかし、よく考えてみたら回避可能です。リアルタイムカーネルを利用しない場合についても全レジスタを保存しなければならないが、実際にユーザーにアセンブリ言語で記述を強制する部分はなく、C 言語だけで割込みハンドラが記述できるものになっています。(ただし RX では一部例外があるので後述します[説明 1])。つまり、#pragma や__attribute__ ((interrupt))などの命令でハンドラ関数はコンパイラが自動でレジスタをすべて保存/復元をしています。そういうことなので､カーネルの ret_int でもコンパイラの拡張命令を使って復元すればいいのです。 ですが、コンパイラの拡張命令は CPU の種類によりさまざまです。しかし、カーネル内に記述したくありません。答えは簡単です。ユーザーハンドラから、カーネルの ret_int を呼び出し、ret_int で直接ディスパッチせず、割込み関数に関数リターンすればいいのです。ユーザーハンドラはカーネルの範疇外のもので、ユーザーが CPU やコンパイラの種類に応じて全レジスタを保存/復元するのですから、そこへ関数リターンすることで、それを利用できます。ただし、利用するのは、RUN 中に割りこまれたタスクがプリエンプトされない場合か、プリエンプト後、再度ディスパッチされる場合のみです。SSP カーネルは極めてシンプルに以下のように実装することで C 言語記述を可能としました。

```
void handler(INTHDR userhandler)
{
        volatile static intptr_t newtskipi;

        intnest++;                      //割り込みネスト数インクリメント
        i_unlock_cpu();                 //割り込み許可
        (*userhandler)();               //ユーザーハンドラ呼び出し
        i_lock_cpu();                   //割り込み不可
```

```
        intnest--;                          //割り込みネスト数デクリメント
        if (intnest == 0)                   //多重割り込み中でない
        {
                if (reqflg !=0)             //スケジュール必要
                {
                        reqflg = 0;
                        run_task(search_schedtsk());
                }
        }
}
```

上記 run_task()は割りこまれたタスクより優先順位の高いタスクがなくなるまでリターンしません。run_task()から戻ってくると一旦ユーザーハンドラにリターンしたのち、割りこまれたタスクの途中から処理を続行します。

ユーザーハンドラは以下のようになります(RX63N の場合)

```
#pragma inline_asm      reg_save
static void
reg_save( void )
{
        pushm   R6-R13
        pushc   fpsw                        ; FPU ステータスレジスタ退避
        mvfacmir5
        shll    #16, r5                     ; ACC 最下位 16bit は 0 とする
        mvfachi r4
        pushm   r4-r5                       ; アキュムレータ退避
}
#pragma inline_asm      reg_load
static void
reg_load( void )
{
        popm    r4-r5                       ; アキュムレータ復帰
        mvtaclo r5                          ; ACC 最下位 16bit は 0 で復帰
        mvtachi r4
        popc    fpsw                        ; FPU ステータスレジスタ復帰
        popm    R6-R13
```

```
}

// CMTU0_CMT0
void Excep_CMTU0_CMT0(void)
{
        reg_save();
        _kernel_handler(CMI0);
        reg_load();
}
```

ユーザーのハンドラーは、

Excep_CMTU0_CMT0()　-> _kernel_handler() --> CMI0()　を呼んでから、ret_int()処理になります。

[説明 1]

上記は GCC での対応の例で、コンパイラが実際に一部のレジスタしか保存されないので、足らないものを保存しています。ルネサス純正コンパイラ(最新版)ならインラインアセンブラ無でコンパイルオプションで対応可能です。

3.待ち状態の追加

待ち状態の追加は、基本的なカーネルのコントロールブロック等を追加します。

特に C 言語記述なので、コンテキストの保存/復帰を基本的に setjmp/longjmp で実装することです。setjmp/longjmp では通常コンテキストは正しく保存されます。

ただし例外があり、割り込みから復帰する場合は、保存復帰が不十分になります。

まず、保存復帰が十分なケースはタスク自らプリエンプトする場合です。

例えば　dly_tsk()や act_tsk()などです。なぜ問題にならないかは、コンパイラが分かっているからというと大雑把な説明ですが、つまり、自ら切り替わる場合は、C のステートメントでの切れ目になり、コンパイラが例えば関数戻り値のレジスタの保存が不要なコード展開をコンパイラがするから関数戻り値のレジスタを保存しなくても問題になりません。一方割りこまれてプリエンプトされ、再びディスパッチする場合は、関数戻り値のレジスタさえ利用中の場合があり、setjmp/longjmp では保存されません。実際のディスパッチャの処理は、以下のような longjmp()を使うものです。

```
void dispatch(intptr_t ipri)
{
        last_ipri = ipri;
```

```
        runtsk_ipri = ipri;
        longjmp(task_ctx[ipri],1);
}
```

割込み時の回避方法として、2 段 jump するものとしました。　つまり、task_ctx[]にタスク別の戻り番地を含むコンテキストを保存しているロジックですが、割り込みから、プリエンプトされる場合は、ret_int 内の出口付近にそのタスクがその状態でそこに戻るように task_ctx[]を保存します。そうすることで、そのタスクが復帰する場合に一旦、ret_int の出口に復帰し、ret_int からリターン、ユーザーハンドラのレジスタ全復帰を経て元のタスクに戻る仕組みです。

4.シュリンク版 SSP カーネルの他のターゲットへの移植方法

機種依存マクロ(関数）以下のものを用意

t_lock_cpu() t_unlock_cpu() ipl_maskClear()

i_lock_cpu()と i_unlock_cpu()は t_と同じで OK

ipl_maskClear()は、割り込みマスクレベルを 0（ユーザーモード）

にするマクロです。

さらに

t_sense_lock() sense_context()

i_sence_lock()は t_と同じで OK

があれば、サービスコール時のエラーチェック一部可能になります。

最悪なくても可能です。

さらに待ち有カーネルの場合には

set_task_stack()　引数をスタックポインタに設定するマクロ

が必要になります。

5.ソースコード

カーネルのみ　https://github.com/alvstakahashi/SHRINK-SSP-KERNEL-ONLY

master ブランチが待ち無/ WAIT-SSP ブランチが待ち有

RaspberryPi 版 https://github.com/alvstakahashi/RPI-SHRINK-SSP-FULL

GR-SAKURA 版 HEW シミュレータ版

https://github.com/alvstakahashi/RX62N-WAIT-SSP-HEWSIM

ターゲットは GR-SAKURA にしていますが、HEW のシミュレータで、実機無で動作確認できます。

6.RaspberryPi への移植サンプル「タミヤラジコン改造スマホリモコンカー」

RaspberryPi のシュリンク版 SSP の移植のサンプルプログラムとして
タミヤのラジコンカーを改造して、スマホリモコンで動作させるサンプル
アプリケーションプログラムを作りました。
オープンソースカンファレンス関西 2015 の TOPPERS ブースでデモしたものです。
内容的には、RC サーボを PWM 制御をするものですが、ステアリングサーボは
マイコンのタイマーハードウェアで対応、リアモーターは周期ハンドラによる
ソフトウェア制御で実現しています。

ソースコード　https://osdn.jp/downloads/users/8/8655/SSP_4tamiya-005.zip
参考動画　https://www.youtube.com/watch?v=3eBIp50r60Y

付録 A

黄色のモジュールが、シュリンク版として残したモジュール

ファイル	場所	モジュール	呼び出し	説明
start.src	arch	_start	_sta_ker	
			_software_init_hook	
			_hardware_init_hook	
			_kernel_istkpt	kernel_cfg.cにて"_kernel_istkpt"の値を決定す
target_support.src	target	_hardware_init_hook		
		_software_init_hook		
startup.c	kernel	sta_ker	target_initialize();	削除
			initialize_object();	kernel_cfg.c
			call_inirtn();	kernel_cfg.c 削除
			start_dispatch();	
		ext_ker		未実装
		exit_kernel		未実装
target_config.c	target	target_initialize	prc_initialize();	ターゲット固有のSIOドライバ
		target_exit	rx600_uart_init その後SIOの初期化	
		target_fput_log		
ファイル	場所	モジュール	呼び出し	説明
prc_config.c	arch	prc_initialize		intnest = 1U; のみ
		prc_terminate		割り込みベクター
		xlog_sys		
		x_config_int		
		default_int_handler		
		default_exc_handler		
rx600_usrt.c	target			SIOドライバ
target_serial.c	target			SIOドライバ
target_timer.c	target			タイマードライバ
banner.c	syssvc			バナー
serial.c	syssvc			SIOドライバ
banner.tf	syssvc			ここでバナーの固定テーブルのテンプレートがあるので削除した
kerbel_cfg.c		_kernel_initialize_object	_kernel_initialize_task();	
			_kernel_initialize_interrupt();	割り込みベクタの初期化 削除
			_kernel_initialize_exception();	例外ベクタの初期化 削除

黄色のモジュールが、シュリンク版として残したモジュール

ファイル	場所	モジュール	呼び出し	説明
task.c	kernel	initialize_task		タスク情報の初期化
		get_ipri_self		
		get_ipri		
		bitmap_search		
		primap_empty		
		primap_serach		
		primap_test		
		primap_set		
		primap_clear		
		swerach_schedtsk		
		test_dormanr		
		make_active		
		run_tsk		
		dispatcher		
ファイル	場所	モジュール	呼び出し	説明
prc_support.src	arch	__kernel_start_dispatch		起動時のディスパッチャ
		__kernel_call_exit_kernel		カーネル出口処理　未実装
		ret_int		割り込みからのディスパッチャ入口
		ret_int_r_rte:		割り込みで割り込みがひとつネストを戻して戻る
		_kernel_interrupt:		割り込みハンドラ
		_kernel_exception		CPU例外 削除
interrupt.c	kernel	initialize_interrupt		割り込み情報のテーブル初期化をしている この辺は手動にするので、削除する方向
		dis_int		ユーザーAPI
		ena_int		ユーザーAPI
exception.c	kernel	initialize_exception		CPU例外のベクタ等の設定